

Chapter 12 Dynamic Memory

The programs we've written so far use variables (objects) that have well-defined lifetimes.

Global objects are allocated at program start-up and destroyed when the program ends. We will talk about global variables today too after dynamic memory.

Local, automatic objects are created and destroyed when the block (between { }) in which they are defined is entered and exited.

Dynamically allocated objects have a lifetime that is independent of where they are created; they exist until they are explicitly freed.

Static memory is allocated on the **stack** and the size is known before the program execution, while **dynamic memory** is allocated on the **heap** during program execution¹.

There is a very nice tool to visualize memory:

<http://pythontutor.com/cpp.html#mode=display>

C++ (g++ 9.3, C++20 + GNU extensions)
([known limitations](#))

```

1 void addOne(int* p){
2     (*p)++;
3 }
4
5 int main() {
6     int *ip = new int(100);
7     addOne(ip);
8     int j = *ip + 2;
9     delete ip;
10    return 0;
11 }
```

[Edit this code](#)

→ line that just executed
→ next line to execute

<< First
< Prev
Next >
Last >>

Step 7 of 10

[Customize visualization](#)

Stack
Heap

The diagram illustrates memory allocation. On the **Stack**, the **main** function has a variable **ip** (pointer) that points to an **int** at memory address **101**. The **addOne(int*)** function also has a parameter **p** (pointer) that points to the same **int** at address **101**. On the **Heap**, there is an **array** of **int** at address **101**.

¹ For more information on stack and heap, read URL: <https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/>

```
void addOne(int* p) {
    (*p)++;
}

int main() {
    int *ip = new int(100);
    addOne(ip);
    int j = *ip + 2;
    delete ip;
    return 0;
}
```

Properly freeing dynamic objects turns out to be a surprisingly rich source of bugs.

2.3.2 The `nullptr`

The `nullptr` is a literal that does not point to any object. We can use it to initialize pointers.

The `nullptr` is a literal that has a special type that can be converted to any other pointer type. Older programs sometimes use a **preprocessor variable** named `NULL`, which the `cstdlib` header defines as 0.

Advise: Uninitialized pointers are a common source of run-time errors. Using an uninitialized pointer almost always results in a run-time crash. However, debugging the resulting crashes can be surprisingly hard.

You can trace the following program at <http://pythontutor.com/cpp.html#mode=display>

```
int main() {
    int j = 3;
    int * i = &j;
    j = 5;
    i = nullptr;

    return 0;
}
```

Allowing variable length arrays is a C99 feature, and is not proper C++ code. This is still allowed in many compilers. To turn this feature off, provide to the compiler, with the following flag: `-pedantic-errors`.

Extra Compiler Flags

-pedantic-errors

Ok

Try it on the following code, and you will see the error of varying array size:

```
#include<iostream>
using namespace std;

int main() {
    int k;
    cout << "Input array size: ";
    cin >> k;
    int br[k];
}
```

Compilation failed due to following error(s).

```
main.cpp:8:7: error: ISO C++ forbids variable length array 'br' [-Wvla]
    8 |     int br[k];
      |         ^^
```

For more about why we need static and dynamic memories, you can read this nice article online².

12.1 Dynamic Memory

In its most elementary form, C++ dynamic memory is managed through a pair of operators:

- **new**, which **allocates, and optionally initializes**, an object in dynamic memory and **returns a pointer** to that object
- **delete**, which **takes a pointer to a dynamic object**, destroys that object, and frees the associated memory.

```
int * p1 = new int;
*p1 = 1001;
int * p2 = new int(1001); // *p2 = 1001
int * p3 = new int(); // value initialized to 0; *p3=0
...
delete p1;
```

² <https://www.geeksforgeeks.org/static-and-dynamic-memory-allocation-in-c/>

```
| delete p2;  
| delete p3;
```

The `new` operator computes the size of the requested memory, and it returns enough memory.

```
| int * p1 = new int;  
| *p1 = 1001;  
  
| delete p1;
```

The variable `p1` is set to point to that memory and must eventually return this memory back to the operating system (OS), an operation called **freeing** the memory. The `delete` operation frees up the memory allocated through `new`.

Until `p1` is freed, the memory that is pointed to will be marked as in-use and will not be given out again by the OS. If you keep allocating memory and never free it, your computer will soon run out of memory.

Dynamic Arrays

This can now be achieved alternatively by the `std::vector<T>` in STL³. Nonetheless, you can dynamically allocate an array of memory using `new` and assign that memory to a pointer:

```
| int *p_numbers = new int[8];
```

Using the array syntax as the argument to `new` tells the compiler how much memory it needs – enough for an 8 element integer array. Now you can use `p_numbers` just as if it pointed to an array. To free the memory, there is a special syntax for the `delete` operator:

```
| delete[] p_numbers;
```

The brackets `[]` tell the compiler that the pointer points to an array of values.

When we delete a pointer to an array, the empty bracket pair is essential: It indicates to the compiler that the pointer addresses the first element of an array of objects. If we omit the brackets when we delete a pointer to an array (or provide them when we delete a pointer to an object), the behavior is undefined.

³When used properly, the raw array using `new` and `delete` will be faster than `vector<T>`.

In-class Exercise 12.1: write a program to dynamically allocate the requested memory for a string array via `new`, change all the values to “C++” and output the contents. Sample run:

```
Enter the size of the string array: 6
The new contents of the string array are:
C++ C++ C++ C++ C++ C++
```

A:

```
#include <iostream>
#include <string>

using namespace std;

int main(){
    cout << "Enter the size of the string array: ";
    int num;
    cin >> num;
    string* parr = new string[num];
    for (unsigned i = 0; i < num; ++i)
        parr[i] = "C++";
    cout << endl;
    cout << "The new contents of the string array are: " << endl;
    for (unsigned i = 0; i < num; ++i)
        cout << parr[i] << " ";
    cout << endl;
    delete[] parr;
    return 0;
}
```

By default, objects allocated by `new`—whether allocated as a single object or in an array—are **default initialized**⁴. We can **value initialize**⁵ the elements in an array by following the size with an empty pair of parentheses.

```
int *pia = new int[10];
int *pia2 = new int[10]();
string *psa = new string[10];
string *psa2 = new string[10]();
```

The following program can be tested, and the default initialization behavior might be seen:

```
| #include<iostream>
```

⁴Default initialized usually means nothing is done; initialized to indeterminate values

URL: https://en.cppreference.com/w/cpp/language/default_initialization

⁵Value initialized usually means numerical types are initialized to 0. URL:

https://en.cppreference.com/w/cpp/language/value_initialization

```
using namespace std;

void m(){
    int *a = new int[5]; // you can fix this by using the next line
    //int *a = new int[5] ();
    cout << a[4] << endl;
    a[4] = 30;
    delete[] a;
}

int main() {
    m();
    m();
    m();
    return 0;
}
```

We can also provide a braced list of element initializers:

```
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// block of ten strings; the first four are initialized from
// the given initializers
// remaining elements are value initialized
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

As when we list initialize an object of built-in array type, the initializers are used to initialize the first elements in the array. If there are fewer initializers than elements, the remaining elements are value initialized.

If there are more initializers than the given size, then the new expression fails and no storage is allocated. In this case, new throws an exception of type `bad_array_new_length`. Like `bad_alloc`, this type is defined in the new header⁶.

We can use an arbitrary expression to determine the number of objects to allocate:

```
// get_size returns the number of elements needed
size_t n = get_size();
int* p = new int[n]; // allocate an array to hold the elements
for (int* q = p; q != p + n; ++q)
    /* process the array */ ;
```

What happens if `get_size` returns 0? The answer is that our code works fine. Calling `new[n]` with `n` equal to 0 is legal even though we cannot create an array variable of size 0:

⁶The library is about memory management URL:

<https://en.cppreference.com/w/cpp/header/new>

```
char arr[0]; // error: cannot define a zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

When we use `new` to allocate an array of size zero, `new` returns a valid, nonzero pointer. That pointer is guaranteed to be distinct from any other pointer returned by `new`. This pointer acts as the off-the-end pointer for a zero-element array.

The memory allocation using `new` and `delete` is the same with objects and classes:

```
#include<iostream>
using namespace std;

class A {
public:
    int a;
};

int main() {
    obj = new A; // default initialized
    cout << "garbage: " << obj->a << endl;
    obj->a = 10;
    delete obj;

    obj = new A(); // value initialized
    cout << "value init: " << obj->a << endl;
    delete obj;
}
```

You might get lucky and see the first value to be 0. But this does not change the fact that the values are undefined when default initialized.

Dynamic memory is problematic because it is surprisingly hard to ensure that we free memory at the right time. Either we forget to free the memory—in which case we have a memory leak—or we free the memory when there are still pointers referring to that memory—in which case we have a pointer that refers to memory that is no longer valid.

The Library for Dynamic Memory Allocation

To make usage of dynamic memory easier (and safer), the library provides smart pointer that manage dynamic objects. A smart pointer acts like a regular pointer with the important exception that **when no longer used it automatically deletes the object to which it points**.

Smart pointers (we will talk about one type here⁷) differ in how they manage their underlying pointers. The `shared_ptr` allows multiple pointers to refer to the same object. To utilize the smart pointers, we need to include the `memory` header.

Smart pointers are templates. When we create a smart pointer, we must supply the type to which the pointer can point:

```
shared_ptr<string> p1;
shared_ptr<list<int>> p2;
```

Dereferencing a smart pointer returns the object to which the pointer points. When we use a smart pointer in a condition, the effect is to test whether the pointer is null:

```
// if p1 is not null, check whether it's the empty string
if (p1 && p1->empty())
    *p1 = "hi";
```

Table 12.1. Operations Common to `shared_ptr` and `unique_ptr`

<code>shared_ptr<T> sp</code>	Null smart pointer that can point to objects of type T.
<code>unique_ptr<T> up</code>	
<code>p</code>	Use <code>p</code> as a condition; true if <code>p</code> points to an object.
<code>*p</code>	Dereference <code>p</code> to get the object to which <code>p</code> points.
<code>p->mem</code>	Synonym for <code>(*p).mem</code> .
<code>p.get()</code>	Returns the pointer in <code>p</code> . Use with caution; the object to which the returned pointer points will disappear when the smart pointer deletes it.
<code>swap(p, q)</code>	Swaps the pointers in <code>p</code> and <code>q</code> .
<code>p.swap(q)</code>	

Table 12.2. Operations Specific to `shared_ptr`

<code>make_shared<T>(args)</code>	Returns a <code>shared_ptr</code> pointing to a dynamically allocated object of type T. Uses <code>args</code> to initialize that object.
<code>shared_ptr<T> p(q)</code>	<code>p</code> is a copy of the <code>shared_ptr</code> <code>q</code> ; increments the count in <code>q</code> . The pointer in <code>q</code> must be convertible to <code>T*</code> (§ 4.11.2, p. 161).
<code>p = q</code>	<code>p</code> and <code>q</code> are <code>shared_ptr</code> s holding pointers that can be converted to one another. Decrements <code>p</code> 's reference count and increments <code>q</code> 's count; deletes <code>p</code> 's existing memory if <code>p</code> 's count goes to 0.
<code>p.unique()</code>	Returns true if <code>p.use_count()</code> is one; false otherwise.
<code>p.use_count()</code>	Returns the number of objects sharing with <code>p</code> ; may be a slow operation, intended primarily for debugging purposes.

⁷ There are also the two smart pointers: `unique_ptr` and `weak_ptr`.

The `make_shared` Function

This function allocates and initializes an object in dynamic memory and returns a `shared_ptr` that points to that object. It is defined in the `memory` header.

When we call `make_shared`, we must specify the type.

```
// shared_ptr that points to an int with value 42
shared_ptr<int> p3 = make_shared<int>(42);
// p4 points to a string with value 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 points to an int that is value initialized to 0
shared_ptr<int> p5 = make_shared<int>();
```

We can also use `auto` to define an object to hold the result of `make_shared`:

```
// p6 points to a dynamically allocated, empty vector<string>
auto p6 = make_shared<vector<string>>();
```

When we copy or assign a `shared_ptr`, each `shared_ptr` keeps track of how many other `shared_ptr`s point to the same object:

```
// object to which p points has one user
auto p = make_shared<int>(42);
// p and q point to the same object, use_count() now returns 2
auto q(p);
```

Once a `shared_ptr`'s counter goes to zero, the `shared_ptr` automatically frees the object:

```
// int to which r points has one user
auto r = make_shared<int>(42);
r = q;
// assign to r, making it point to a different address
// increase the use count for the object to which q points
// reduce the use count of the object to which r had
// pointed freed
```

When the last `shared_ptr` pointing to an object is destroyed, the `shared_ptr` class automatically destroys the object to which that `shared_ptr` points: memory being freed.

You can trace the pointer counts of the following program.

```
#include <memory>
#include <iostream>
```

```

using namespace std;

void m(shared_ptr<int>& a){
    cout << "2: " << a.use_count() << endl;
    shared_ptr<int> b(a);
    cout << "3: " << a.use_count() << endl;
    cout << "4: " << a.use_count() << endl;
    return;
}

int main() {
    shared_ptr<int> j;
    {
        shared_ptr<int> i;
        i = make_shared<int>(5);
        cout << "1: " << i.use_count() << endl;
        m(i);
        j = i;
        cout << "5: " << i.use_count() << endl;
    }
    cout << "6: " << j.use_count() << endl;
    j = nullptr; // destroy object pointed by shared_ptr j
    cout << "7: " << j.use_count() << endl;
}

```

We can also have `shared_ptr` for arrays. You can try out the following code.

```

#include <iostream>
#include <fstream>
#include <memory>
#include <unistd.h>

using namespace std;

/* https://gist.github.com/thirdwing/da4621eb163a886a03c5 */
void process_mem_usage();

int main(){
    cout << "Before doing anything. " << endl;
    process_mem_usage();

    srand(0);
    // needs C++ 17
    shared_ptr<int[]> my_array( new int[1000000000]);
    // prior C++ 17, you need to provide the delete setup:
    //shared_ptr<int> my_array(
    // new int[1000000000], default_delete<int[]>() );
    cout << "Allocating memory now! " << endl;
    for(int i = 0; i < 1000000000; ++i)
        my_array.get()[i] = rand()%100;
    process_mem_usage();

    // my_array now points to nowhere,
    // and the large array is released

```

```

    cout << "releasing memory now!" << endl;
    my_array = nullptr;
    process_mem_usage();
}

void process_mem_usage(){
    double vm_usage = 0.0;
    double resident_set = 0.0;

    // the two fields we want
    unsigned long vsize;
    long rss;
    {
        string ignore;
        ifstream ifs("/proc/self/stat", ios_base::in);
        ifs >> ignore >> ignore >> ignore >> ignore >> ignore >> ignore >>
ignore >> ignore >> ignore >> ignore >> ignore
        >> ignore >> ignore >> ignore >> ignore >> ignore >>
ignore >> ignore >> ignore >> ignore >> ignore
        >> ignore >> ignore >> vsize >> rss;
    }

    long page_size_kb = sysconf(_SC_PAGE_SIZE) / 1024;
    vm_usage = vsize / 1024.0;
    resident_set = rss * page_size_kb;
    cout << "VM: " << vm_usage << "; RSS: " << resident_set <<
endl;
}

```

Programs tend to use dynamic memory for one of three purposes:

1. They don't know how many objects they'll need.
2. They don't know the precise type of the objects they need.
3. They want to share data between several objects.

The container classes are an example of classes that use dynamic memory.

Global variables

Global variables are defined outside of functions. They are created when the program starts, and destroyed when it ends⁸. This is called **static duration**.

The scope of global variables begins at the point where they are defined and lasts until the end of the file/program.

When uninitialized, local variables are default initialized, while **global variables are zero-initialized**.

⁸ Global variables <https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/>

```
#include<iostream>
using namespace std;

int i; // global variable

void m(){
    i++;
    cout << "in m(), i = " << i << endl;
}

int main() {
    cout << "in main(), i = " << i << endl;
    m();

    int i = 5; // local variable will now shadow global variable
    cout << "local i = " << ::i << endl;
    cout << "get global i through :: " << ::i << endl;
}
```

Global variables across many .cpp files⁹ (same copy: same value)

In the following example, you will have multiple .cpp files sharing the same variable.

The **extern** keyword is used before a variable to inform the compiler that this variable is declared somewhere else.

source.h

```
#ifndef SOURCE_H_
#define SOURCE_H_

#include <iostream>

extern int global;
int function();

#endif
```

source1.cpp

```
#include "source.h"

using namespace std;

int global = 42; // declared here

int main(){
    cout << function();
}
```

⁹ Global visibility in different .cpp files <https://stackoverflow.com/questions/3627941/global-variable-within-multiple-files>

```
| }
```

source2.cpp

```
| #include "source.h"  
  
| using namespace std;  
  
| int function(){  
|     if(global==42)  
|         return 42;  
|     return 0;  
| }
```

Static global variables in many .cpp files (separate copies, can have different values)

m.h

```
| #ifndef M_H_  
| #define M_H_  
  
| #include <iostream>  
  
| void m();  
  
| #endif
```

m.cpp

```
| #include "m.h"  
  
| using namespace std;  
  
| static int i; // separate copy of global variable  
  
| void m(){  
|     cout << "in m(), i = " << i << endl;  
| }
```

main.cpp

```
| #include "m.h"  
  
| using namespace std;  
  
| static int i; // separate copy of global variable  
  
| int main() {  
|     cout << "in main(), i = " << ++i << endl;  
|     m();  
| }
```

The above implementation is an example of **static global variables**. Variables declared as static at the top level of a source file (outside any function definitions) are only visible throughout that file¹⁰.

Static variables in functions

Static local variables: Variables declared as static inside a function are statically allocated, thereby keeping their memory cell throughout all program execution, while also having the same scope of visibility as automatic local variables. Hence whatever values the function puts into its static local variables during one call will remain the same whenever the function is called again¹⁰. **Static variables are zero-initialized by default**¹¹.

```
#include <iostream>

using namespace std;

int factorial(int n){
    static int val; // for the frist execution, zero initilizaed
    cout << ++val << endl;
    if(n >= 1 ) return n * factorial(n - 1);
    else if(n == 0) return 1;
    else return -1;
}

int main(){
    cout << factorial(5);
}
```

7.6 static Class Member

The *static class members* are data and functions that are associated with the class itself. They are not specific to any object. In contrast, the members associated with the object are called *instance members*.

Now consider a class Fred

```
class Fred {
public:
    static void f();// Member function associated with the class
    void g();// Member function associated with an object of the class.
protected:
    static int x; // Data member associated with the class
    int y; // Data member associated with an individual object of the class
```

¹⁰ Static global and static local variables <https://www.faceprep.in/c/difference-static-and-global-variables/>

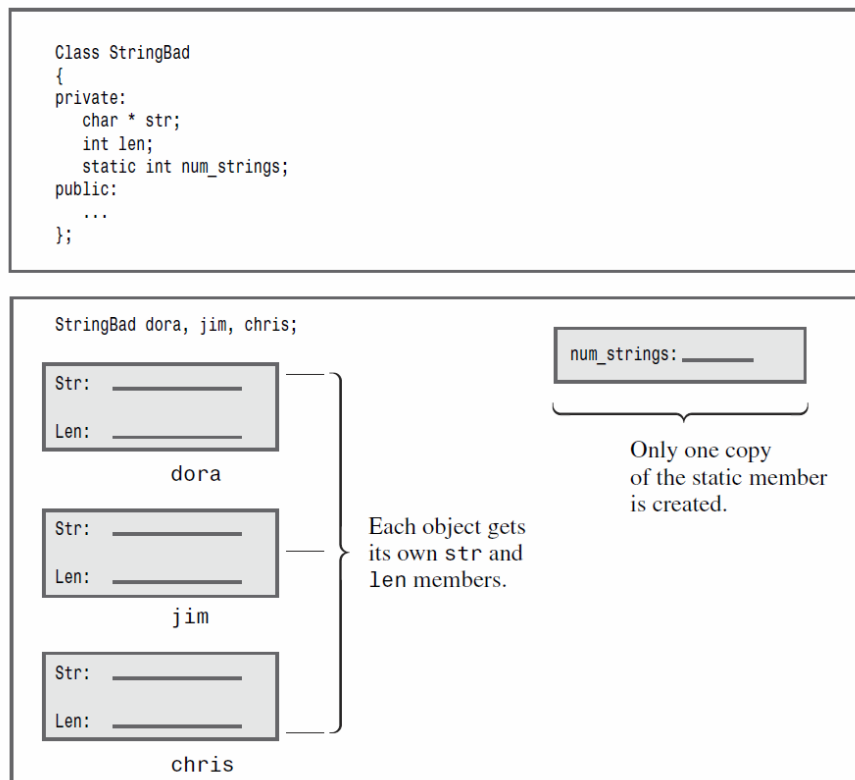
¹¹ Initialization of static variables https://en.cppreference.com/w/cpp/language/zero_initialization

```
| };
```

The class has a static data member x and an instance data member y . There is only one copy of `Fred::x` regardless of how many `Fred` objects are created (even with no `Fred` objects). There is one y per `Fred` object.

Thus, x is said to be associated with the class and y is said to be associated with an individual object of the class. Similarly, class `Fred` has a static member function `f()` and an instance member function `g()`.

The following figure is another example.



Example: a bank account class `Account` need a static data member to represent the current prime interest rate. In this case, we'd want to associate the rate with the class, not with each individual object. From an efficiency standpoint, there'd be no reason for each object to store the rate, because all objects have the same rate. Much more importantly, if the rate changes, we'd want each object to all use the new value.

```
| class Account {
| public:
|     void calculate() { amount += amount * interestRate; }
|     static double rate() { return interestRate; } //get
```

```

    static void rate(double); //set
private:
    std::string owner;
    double amount;
    static double interestRate; // static member data
};

```

The static members of a class exist outside any object. Objects do not contain data associated with static data members. Thus, each Account object will contain two data members—owner and amount. There is only one interestRate that will be shared by all Account objects.

Similarly, static member functions are not bound to any object; they do not have a this pointer. As a result, static member functions may not be declared as const, and we may not refer to this in the body of a static member. This restriction applies both to explicit uses of this and to implicit uses of this by calling a non-static member.

Access Class static Members

We can access a static member directly through the scope operator:

```

double r;
r = Account::rate(); // access static member via scope operator

```

Even though static members are not part of the objects of its class, we can use an object, reference, or pointer of the class type to access a static member:

```

Account ac1;
Account& ac2 = ac1;
Account* ac3 = &ac1;
// equivalent ways to call the static member function rate()
double r;
r = ac1.rate(); // through an Account object
r = ac2.rate(); // through an Account reference
r = ac3->rate(); // through a pointer to an Account object

```

Other Account class member functions can use static members directly, without the scope operator:

```

class Account {
public:
    void calculate() { amount += amount * interestRate; }
private:
    static double interestRate;
// remaining members as before

```



```
| };
```

Define static Members

As with any other member function, we can **define** a static member function inside or outside of the class body.

Because static data members are not part of individual objects of the class type, they are not defined when we create objects of the class. As a result, they are not initialized by the class' constructors.

For static data members and static member functions, we typically define them in the corresponding cpp source file.

When we define a static member outside the class, **we do NOT repeat the static keyword**. The following are definitions of the Account class in the Account.cpp file:

```
| #include "Account.h"
| double Account::interestRate = 0.02;
|
| void Account::rate(double r) {
|     interestRate = r;
| }
```

Moreover, we may not initialize a static member inside the class: we define and initialize each static data member outside the class body, and outside any function. Once they are defined, they continue to exist until the program completes. Like any other object, a static data member may be defined only once (in one of the .cpp file).

If you still remember the compilation process, each .cpp file forms its own object file, and then the linker will link them together. If the static data members are defined in the header, any .cpp including that header will have its own definition of the static data members. Which copy should we use then? As a result, there is no confusion when you have only one definition of the static data member (in one of the .cpp file).

Example: let us put together an Account class and illustrate its usage of static class member.

Account.h

```
| #ifndef ACCOUNT_H
| #define ACCOUNT_H
| #include <string>
```

```

class Account {
public:
    Account() = default;
    Account(std::string name, double initAmount) :
        owner(name), amount(initAmount){ }
    void calculate() { amount += amount * interestRate; }
    static double rate() { return interestRate; } //get
    static void rate(double); //set
    double getAmount() { return amount; }
    std::string getOwner(){ return owner; }
private:
    std::string owner = "Anonymous";
    double amount = 0.0;
    static double interestRate;
};
#endif

```

Account.cpp

```

#include "Account.h"

double Account::interestRate = 0.04; // static var's definition

void Account::rate(double r) {
    interestRate = r; // static var's reassignment
}

```

mainDeposit.cpp

```

#include "Account.h"
#include <iostream>
#include <vector>

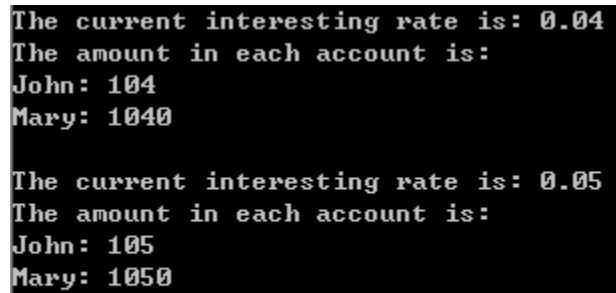
using namespace std;

int main(){
    Account a = Account("John", 100);
    Account b = Account("Mary", 1000);
    vector<Account> coll = {a , b};
    cout << "The current interesting rate is: " << Account::rate()
        << endl;
    cout << "The amount in each account is: " << endl;
    for (auto e : coll){ // copy and will not change e
        e.calculate();
        cout << e.getOwner() << ": " << e.getAmount() << endl;
    }
    cout << endl;

    Account::rate(0.05);
    cout << "The current interesting rate is: " << Account::rate()
        << endl;
    cout << "The amount in each account is: " << endl;
    for (auto e : coll){ // copy and will not change e
        e.calculate();
        cout << e.getOwner() << ": " << e.getAmount() << endl;
    }
}

```

```
    cout << endl;  
  
    return 0;  
}
```



```
The current interesting rate is: 0.04  
The amount in each account is:  
John: 104  
Mary: 1040  
  
The current interesting rate is: 0.05  
The amount in each account is:  
John: 105  
Mary: 1050
```

Remark: static member function vs. ordinary member function

An ordinary member function declaration of a class specifies **three logically distinct things**:

1. The function can access the private part of the class declaration.
2. The function is in the scope of the class.
3. The function must be invoked from an object (or the function has a **this** pointer)

By declaring a member function `static`, we give it the first two properties only. By declaring a function `friend`, we give it the first property only.